NASA Technical Memorandum 4251

Test and Evaluation of the Generalized Gate Logic System Simulator

Paul S. Miner

APRIL 1991

NASA Technical Memorandum 4251

Test and Evaluation of the Generalized Gate Logic System Simulator

Paul S. Miner
Langley Research Center
Hampton, Virginia

National Aeronautics and
Space Administration
Office of Management
Scientific and Technical
Information Division

1991

Contents

Symbols

D_1	faults detected after the first program iteration of LINCON (including those in local memory)
$D_{1\rightarrow 8}$	faults detected in any of the first eight program iterations of LINCON (local random access memory (RAM) only)
D_o	faults visible at an output after the first program iteration of LINCON
D_Σ	faults detectable anywhere in RAM
D_o/D_Σ	proportion of detectable faults visible at an output after the first program iteration
D_o/F	proportion of faults detectable at an output port after the first program iteration
D_o/F_α	proportion of distinguishable faults detectable at an output port after the first program iteration
D_Σ/F	proportion of simulated faults detectable anywhere in RAM
D_Σ/F_α	proportion of distinguishable faults detectable anywhere in RAM
F	total number of simulated faults (same for each run), 1741
F_α	number of simulated faults corrected for known indistinguishable faults, 1558
MATMUL$\bigcup$Run$_1$	faults detected by MATMUL or by LINCON run 1
MATMUL$\bigcup(\bigcup_{j=1}^{14}$ Run$_j)$	faults detected by MATMUL or by any of the 14 LINCON runs
MATMUL$\bigcap$Run$_1$	faults detected by MATMUL and by LINCON run number 1
MATMUL$\bigcap(\bigcap_{j=1}^{14}Run_j)$	faults detected by MATMUL and by all 14 LINCON runs
N_1	number of faults that failed to produce an answer in the first program iteration of LINCON
N_1/D_o	proportion of faults detected in the first iteration that failed to produce an answer
N_1/F	proportion of faults that failed to produce an answer in the first program iteration
N_1/F_α	proportion of distinguishable faults that failed to produce an answer in the first program iteration
N_Σ	number of faults that failed to produce an answer for all program iterations
N_Σ/D_Σ	proportion to detectable faults that failed to produce an answer for all program iterations
N_Σ/F	proportion of faults that failed to produce an answer for all program iterations
N_Σ/F_α	proportion of distinguishable faults that failed to produce an answer for all program iterations

$\bar{\delta}$	coincident error as defined in Swern et al. (ref. 6); given that two faults (one latent) exist in distinct redundant channels of a fault-tolerant digital system, $\bar{\delta}$ is the probability that they produce identical errors
$\bigcup_{j=1}^{14} \mathrm{Run}_j$	faults detected in any of the 14 LINCON runs
$\bigcap_{j=1}^{14} \mathrm{Run}_j$	faults detected in all 14 LINCON runs

Introduction

Focus of This Study

This paper discusses the results of the initial testing of the Generalized Gate Logic System Simulator (GGLOSS). The simulator is a special-purpose fault simulator designed to assist in the analysis of the effects of random hardware failures on fault-tolerant digital computer systems. The testing of the simulator covers two main areas. First, the simulation results are compared with data obtained by monitoring the behavior of hardware. The circuit used for these comparisons is an incomplete microprocessor design based upon the MIL-STD-1750A Instruction Set Architecture. In the second area of testing, current simulation results are compared with experimental data obtained using precursors of the current tool. In each case, a portion of the earlier experiment is confirmed. The new results are then viewed from a different perspective in order to evaluate the usefulness of this simulation strategy.

The structure of the report is as follows. The remainder of this introductory section gives a brief historical perspective of the simulator, a description of the salient features of the GGLOSS simulation strategy, and a description of the microprocessor design. The following section describes the results of comparing the simulation results with data obtained from the laboratory prototype. The final section consists of a comparison of current data with the results from two earlier fault simulation experiments. The first of these is an attempt to estimate the fault coverage of a comparison monitoring fault-tolerant system. The second study attempted to estimate the percentage of coexisting faults that can defeat a comparison monitoring system. In each case, a portion of the earlier study is recreated, and then a different interpretation of the results is given.

Historical Background

There have been a series of studies sponsored by Langley Research Center that have explored the dynamics of gate-level fault behavior in fault-tolerant digital computer systems. In a 1978 study addressing the use of a comparator/voter as a means for detecting faults, Nagel (ref. 1) reported that, for the six sample programs, only $\approx$50 percent of the injected faults produced observable errors after eight program iterations. If the comparator/voter were the only means of fault detection, this would allow multiple faults to accumulate in redundant channels, creating the potential for defeating redundancy management logic. McGough and Swern (refs. 2 and 3) then performed a series of gate-level simulations of a Bendix BDX-930 "avionic mini processor" in order to corroborate Nagel's results by using a realistic digital avionic system. That study measured similar detection probabilities for the six algorithms used by Nagel. In addition, a three-axis flight control computation was simulated. Once again, a significant number of the simulated faults failed to produce an observable error. The BDX-930 simulation was also used to demonstrate a methodology for designing and validating built-in self-test routines. In 1982, McGough performed a feasibility study to identify the salient features of the BDX-930 Gate Logic Software Simulator (BGLOSS) and to determine if a generalized simulator could be developed (ref. 4). The feasibility study concluded that a generalized version could be written, and a prototype generalized simulator was developed. This simulator was called the Interim Generalized GLOSS (IGGLOSS) (ref. 5). Due to limitations in the original IGGLOSS, Langley opted to develop a production grade version of the simulator (GGLOSS). Concurrently, Swern et al. (ref. 6) used an extended version of IGGLOSS (called S-GGLOSS) to simulate a simple 300-gate processor in order to estimate the probability of coincident error in a redundant computing system. The ultimate intent of these studies was to provide some means to estimate fault coverage for the reliability analysis of fault-tolerant digital computer systems.

Description of Simulator

GGLOSS was designed specifically to be a high-speed fault simulator. As such, it lacks features such as circuit timing analysis and multivalued logic, which are common in commercially available design verification simulators. It depends upon the assumption that the simulated circuit is a verified design. Furthermore, since it was designed to be able to simulate processing elements executing application programs, a key development issue was simulation speed. To achieve this, GGLOSS is limited to 2-value logic, which allows the parallel simulation of 32 copies of the circuit on a VAX host. GGLOSS maintains 1 unfaulted copy of the circuit for easy comparison, while allowing the user to inject faults in any of the other 31 copies. The user has the option of injecting single or multiple faults in each of the 31 faultable copies of the circuit. Injected faults may be permanent or intermittent. GGLOSS uses bit masking to inject stuck-at-1, stuck-at-0, and invert faults at any input or output node of any gate in the circuit. A user can monitor the propagation of a fault at any point in the circuit because any location within the simulated circuit can be designated as a test point. GGLOSS compiles the circuit from a netlist description into an internal representation

of primitive functions that are evaluated in an invariant, predetermined order. This eliminates much of the overhead required by an event-driven simulator. The compiled circuit representation implements a combination of zero-delay and unit-delay simulation techniques in order to model combinatorial and sequential circuit elements, respectively. This set of characteristics gives GGLOSS the ability to simulate approximately 10^6 gate evaluations per MicroVAX II cpu-second, while allowing the user the ability to monitor the effects of the simulated faults. In order to simulate a large number of faults, GGLOSS allows the creation of several independent simulations, each consisting of 31 different fault scenarios. These independent simulations can be easily distributed among the nodes of a local area network, achieving performance gains nearly linear with respect to the number of available nodes.

Description of Microprocessor Design

The circuit used in the initial evaluation of GGLOSS is a self-testing microprocessor design based upon the MIL-STD-1750A Instruction Set Architecture (ISA). Reasons for selecting this circuit include the availability of gate-level schematics, documented microcode, and a laboratory prototype circuit. The laboratory prototype implementation was constructed using special chips that allow for the gate-level injection of faults. This feature provided a means for comparing the results of fault simulations in GGLOSS with those obtained by injecting faults in the hardware. The laboratory prototype hardware and documentation were delivered "as is" at the end of the second stage of a three-stage project, and the third stage was not funded. Thus, comments concerning the lack of features within the processor do not imply a criticism of the design, but rather a recognition of the difficulties encountered when working with an unfinished project.

Among the limitations of the hardware design was the lack of a significant portion of microcode. There were no branch instructions, no single precision integer compare, no floating-point operations, no stack operations, and no subroutine calls.[1] Furthermore, the interrupt logic, while present, was not functional. There was a surplus of unused bits in the microcontrol store, but none of these had been assigned to the necessary control signals for the interrupt hardware. While these limitations hampered the simulation effort, it was still possible to use this circuit as a means of testing the simulator. One caveat should be stressed as a result of these limitations; that is, while useful information was gained about the simulator, it is not reasonable to treat the results obtained as typical of production microprocessors. Neither should the results be construed as being relevant to any commercially available MIL-STD-1750A ISA microprocessor, since the processor in question does not meet the full Notice 1 specification. Henceforth, the processor used in this study will be referred to as the "SS-1750A," since it implements a subset of the MIL-STD-1750A Notice 1 specification.

The laboratory prototype design also had features useful to this study. The microcode was stored in a writable control store and thus was easily modified through control of the PC host. Furthermore, the design was implemented using custom SSI (small scale integration) chips, making many of the locations in the design readily accessible to logic analyzer probes. These custom chips allowed for simple injection of faults into the combinatorial logic of the arithmetic logic unit (ALU).

Comparison Between Simulation and Hardware

Unfaulted Testing of Simulator

The initial testing of GGLOSS was performed using partial schematics of the SS-1750A prior to delivery of the laboratory prototype hardware. The schematics had been developed using a computer-aided design (CAD) system, so a machine readable circuit description (netlist) could be generated automatically. Individual netlists were generated of various functional components, including the arithmetic logic unit (ALU), microsequencer, general purpose register file, and the I/O (input/output) registers. After several iterations of modifications to the schematics[2] and to the part mapping definitions for the GGLOSS Circuit Ingest environment,[3] a valid internal representation of each of these subcircuits was obtained. These were each simulated for a few test cases in order to check for errors in the netlists. Ultimately, the schematics were combined, and a netlist corresponding to the usable portion of the design was produced.[4]

The effort required to verify correct unfaulted simulation of the microprocessor was compounded by

[1] Microcode for some of the missing instructions was developed by the author in order to perform this study.

[2] For example, the most common modification was the addition of part attributes to the symbols in order to generate a valid netlist.

[3] The Circuit Ingest environment consists of the set of programs that map external circuit descriptions to the appropriate internal primitive representations.

[4] The interrupt logic was not included in the simulation.

the fact that the documentation of the processor was incomplete, there were errors in the schematic, and the simulator was still being developed. Thus, any discrepancy between execution of the simulator and the prototype hardware could be caused by an error in any of these areas. Several differences were found between the behavior of the initial simulation attempt and the behavior of the hardware. A few discrepancies were traced to flaws in the implementation of the simulator. These flaws were immediately corrected by the GGLOSS development team. Many of the discrepancies were caused by misinterpretation of the processor documentation, while some were due to incorrect or incomplete documentation. Eventually a working simulation of the processor was obtained.

Self-Test Fault Simulation

Description of self-test. In the initial phase of the GGLOSS evaluation, the self-test mechanism of the SS-1750A processor was exercised. The self-test hardware for the data path of the processor consists of a linear feedback shift register (LFSR) for generating pseudorandom test patterns, and a multiple input signature register (MISR) for compressing the resultant signature.[5] The data path test is controlled by the microcode. Two different algorithms are used for testing registers, with the results of each test shifted into the MISR. The data path test also checks the ALU logic by using the LFSR to generate input patterns. For each test pattern, several ALU functions are exercised, and all intermediate results are shifted into the MISR.

On the SS-1750A there are two modes for execution of the self-test microcode. The first mode is used to generate a "good-machine" signature, which is required to evaluate the results of subsequent tests. In this mode, the microcode loop that generates the pseudorandom test patterns and exercises the ALU logic is repeated for exactly 1024 patterns. After the loop terminates, the contents of the MISR are stored into the good-machine signature register. This is the only situation in which it is possible to write into this register, at any other time it can only be read. Thus the first mode consists of generating the good-machine signature necessary for comparison during subsequent self-test execution.

The second mode consists of testing for the presence of faults. In this mode, the test loop is repeated until the contents of the MISR are equal to the previously generated good-machine signature. In other words, the loop is now nonterminating if the circuit fails the test. However, it is possible for a fault to alter the execution of the test such that a valid signature is generated in a different number of iterations than required to produce a "good" signature. In this case, the fault is actively causing erroneous behavior, but is undetected by the test. The fact that a failed test is nonterminating is unfortunate because the only way to recognize that a component has failed is if it does not claim to be good within a fixed time interval.

Results and analysis. Once a few discrepancies caused by errors in the processor documentation were resolved, the good-machine signature generated by the GGLOSS simulation was identical to the signature generated by the SS-1750A hardware. It was then possible to make comparisons of the faulted behavior. The laboratory prototype processor allows for the injection of 1312 distinct stuck-at faults in the gates of the arithmetic logic unit. The faults can be inserted only in the combinatorial logic. Table I shows the class of faults injected for each combinatorial gate type. All these faults were injected in the hardware prototype. For each fault, the self-test was executed for a fixed number of clock cycles, and the results contained in the MISR at the end of that interval were saved. Additionally, the contents of the MISR were compared with the previously generated good-machine signature in order to measure coverage of the test.

Table I. Fault Set—Self-Test

Gate	Input pins	Output pins
And	Stuck-at-1	Stuck-at-0
Or	Stuck-at-0	Stuck-at-1
Nand	Stuck-at-1	Stuck-at-1
Nor	Stuck-at-0	Stuck-at-0

In the GGLOSS simulation, it was not necessary to initialize the good-machine signature register, as the unfaulted scenario[6] would always generate the appropriate signature value in time for the intended comparison. By not initializing the good-machine signature, the termination condition for the self-test loop in the GGLOSS simulation was different from that on the hardware, but only in the case where the fault caused a good-machine signature at an inappropriate time. Remember that the structure of the self-test is such that it is possible for a fault to

[5] See P. K. Lala's text for discussion of how to implement a LFSR/MISR combination (ref. 7, p. 229–30).

[6] Remember that in GGLOSS there is always one fault-free copy of the circuit maintained.

generate a valid signature, which causes early exit from the test. Setting up the simulation in this manner not only allowed for reduced simulation time, but raised the possibility of identifying faults that defeat the self-test algorithm.

Of the 1312 stuck-at faults, 1300 were detected by this test, both in the hardware prototype and in the GGLOSS simulation. However, when the signatures generated on the prototype hardware were compared with those generated by the GGLOSS simulation, 9 of the 1300 detected faults had signatures that disagreed. Either there was an error in the simulation or these nine faults actually defeated the test. On the hardware, the presence of these faults had been observed by the PC host that was monitoring the test, but they had not actually been detected by the self-test. Subsequent executions on the laboratory prototype demonstrated that these 9 faults exited the self-test with a good-machine status prior to the 1024th pass through the test loop. Even though the SS-1750A documentation recorded these nine faults as being detected by the self-test, the results of this study indicate that they were undetected by the test and returned control to the processor.[7] Thus, the GGLOSS simulation revealed a previously undocumented error in the design of the self-test.

Comparison With Previous Experiments

Comparison Monitoring Coverage

Description. A suitable reference point for determining the applicability of GGLOSS is an investigation of the results of the BGLOSS simulation of the BDX-930 (refs. 2 and 3). It will be shown that results generated from the simulation of the SS-1750A correspond closely to those produced in the BDX-930 study. However, the results will also be viewed from a different perspective. Both the Nagel study and the BDX-930 studies demonstrate that comparison monitoring systems[8] fail to detect all possible faults (refs. 1, 2, and 3). Similar results can be shown using the SS-1750A.

There may exist faults that will remain undetected by a comparison monitor and then subsequently exhibit malicious behavior.[9] Experiments to date have not provided a better understanding of these potentially malicious latent faults. All that has been determined is that the majority of stuck-at faults do not exhibit malicious behavior. In other words, these studies have provided a better understanding of the behavior of nonlatent or short-term latent faults. None of the studies were able to determine if any of the undetected faults (possibly long-term latents) could exhibit malicious behavior.

There were two simple 1750A programs used in this part of the study. The code is given in the appendix. The first implements the LINCON[10] (refs. 2 and 3) algorithm from the BGLOSS simulations of the BDX-930. The second is a matrix multiplication (MATMUL) routine that squares a 2×2 matrix of floating-point data. Eight program iterations of LINCON require approximately 20 000 clock cycles to complete on the SS-1750A processor. MATMUL requires approximately 10 000 clock cycles to complete (worst-case estimate). Assuming a 10 MHz clock rate for the processor, these programs require 2 ms and 1 ms of real time, respectively, to complete. There were 14 sets of input data generated randomly for the LINCON program. Each input set was selected in accordance with the criteria used in the BDX-930 study. Only one set of data was required for the comparison monitoring experiment. The multiple sets of data were required for the section on coincident error. The MATMUL program was executed using a single set of floating-point data consisting of all nonzero entries. Both positive and negative values were used in order to fully exercise the floating-point operations. Using these two programs, many of the capabilities of GGLOSS were exercised.

The set of faults (F) for the SS-1750A simulation were selected from the microsequencer and the ALU. Faults were injected in the combinatorial logic only and were selected according to the criteria presented in table I. A total of 1741 faults was selected, including the 1312 used in the evaluation of the self-test. These faults were simulated for each of the 14 LINCON executions, as well as for the execution of MATMUL.

[7] With the addition of a counter these faults could also be detected, as one could ignore any good-machine signals until after execution for a fixed number of clock cycles.

[8] A comparison monitoring system is one that uses tests for equivalence between redundant systems in order to detect a faulty channel.

[9] For example, consider a fault that can only exhibit erroneous behavior when the system attempts to reconfigure. The effects of such a fault cannot appear until a system is attempting to recover from a second fault. Such a fault could prevent the system from reconfiguring, even if sufficient hardware was available.

[10] A simple program performing arithmetic operations on integer data which is similar in structure to control programs. This program was chosen because, of all the programs simple enough to implement on the SS-1750A, the observed fault behavior for this program was closest to that of the flight control computation simulated in the BDX-930 study.

Results. Table II presents initial detection results from the SS-1750A simulation for each of the 14 distinct LINCON runs. For this table, D_i represents the number of faults detected in the ith iteration that were not detected in any previous iteration. The detectability criteria for this table include only those errors observable at the single output of the program or at the memory location for internal feedback data. Errors observable elsewhere were not counted. Column $D_{1\rightarrow 8}$ represents those faults detected in any of the eight program iterations. As was shown in the previous studies, the majority of faults detected were detected in the first iteration. Also, while a majority of faults were detected, a significant percentage remained undetected after eight iterations of the program. These results are similar to those presented for the BDX-930 simulation of the LINCON algorithm. In that study (refs. 2 and 3), the LINCON program was executed for eight iterations using a single set of randomly generated input data. Of 807 injected gate-level[11] faults, 547 were detected for a coverage estimate of 0.678. Of the 547 detected faults, 529 caused an error in the first iteration of the program with the remaining faults detected in the 2nd through 8th iterations.

Table II. Detection by Iteration—LINCON

LINCON	$D_{1\rightarrow 8}$	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8
Run_1	1331	1227	54	16	15	9	8	2	0
Run_2	1302	1130	76	37	17	10	16	16	0
Run_3	1319	1072	134	30	21	39	3	8	12
Run_4	1321	1178	71	21	18	23	8	1	1
Run_5	1311	1211	53	28	9	4	0	6	0
Run_6	1284	1176	47	28	15	10	8	0	0
Run_7	1292	1151	83	20	17	17	1	3	0
Run_8	1320	1090	143	28	20	25	12	2	0
Run_9	1306	1184	49	35	6	12	10	6	4
Run_{10}	1328	1198	57	41	15	12	3	2	0
Run_{11}	1310	1074	138	52	9	8	15	8	6
Run_{12}	1325	1168	71	12	22	15	17	19	1
Run_{13}	1289	1191	36	30	11	3	16	2	0
Run_{14}	1328	1250	36	24	17	0	0	0	1

Departing from the earlier results, table III gives various coverage factors for the 14 executions of the LINCON program. The criteria used for detection in this table are slightly different from those used in table II; D_Σ represents the set of faults that corrupted any location of memory (e.g., the entire contents of RAM were compared) and D_o represents only those errors that would be visible at an output port after the first iteration of the program. Except for runs 6 and 7, the set of faults detectable anywhere were detectable in the memory local to the process. Only in LINCON runs 6 and 7 did a fault corrupt memory outside the local memory space of the program. The final two rows in table III give results that explore detection across multiple runs. Of the 1342 faults that were detected by at least one run ($\bigcup_{j=1}^{14} \text{Run}_j$), 1306 had at least one externally visible detection in the first iteration. Similarly 1267 faults were detectable in every run ($\bigcap_{j=1}^{14} \text{Run}_j$).

Table III. Coverage Factors—LINCON

LINCON	D_Σ	D_o	D_Σ/F	D_Σ/F_α	D_o/F	D_o/F_α	D_o/D_Σ
Run_1	1331	1227	0.765	0.854	0.705	0.788	0.922
Run_2	1302	1074	.748	.836	.617	.689	.823
Run_3	1319	1029	.758	.847	.591	.660	.780
Run_4	1321	1103	.759	.848	.634	.708	.835
Run_5	1311	1182	.753	.841	.679	.759	.902
Run_6	1290	1097	.741	.828	.630	.704	.850
Run_7	1298	1110	.746	.833	.638	.712	.855
Run_8	1320	1056	.758	.847	.607	.678	.800
Run_9	1306	1184	.750	.838	.680	.760	.907
Run_{10}	1328	1183	.763	.852	.679	.759	.891
Run_{11}	1310	1062	.752	.841	.610	.682	.811
Run_{12}	1325	1155	.761	.850	.663	.741	.872
Run_{13}	1289	1160	.743	.830	.666	.745	.900
Run_{14}	1328	1249	.763	.852	.717	.802	.941
$\bigcup_{j=1}^{14} \text{Run}_j$	1342	1306	0.771	0.861	0.750	0.838	0.973
$\bigcap_{j=1}^{14} \text{Run}_j$	1267	914	.728	.813	.525	.587	.721

As in the BDX-930 study, an attempt was made to remove the set of indistinguishable[12] faults from consideration in the coverage factors; F_α consists of the 1558 faults that were not identified as indistinguishable. Of the 1741 simulated faults F, 204 never produced observable erroneous behavior (determined by combining results from the self-test and the LINCON and MATMUL simulations). Thus, 1537 of the faults in F are clearly detectable and therefore in F_α. The remaining 204 faults were analyzed to determine why they were not detected. Faults were identified as undetectable based upon analysis of the circuit

[11] Although the BDX-930 study included PROM bit faults, they are excluded from this summary, since this study did not consider memory faults.

[12] "A fault that has no affect [sic] on the computational process is indistinguishable .... a distinguishable fault has the property that there exists a software program the output of which differs from that of the same program executed by an identical but non-faulted processor." (ref. 2, p. 16).

and microcode. Since the circuit being simulated was still in the design phase, there was a significant proportion of unused logic present for anticipated changes in the design. Furthermore, given the nature of the microcode, there were a number of faults that could never be detected with the current microcode but would possibly be detectable using a different implementation. Additionally, some faults were identified as being in redundant logic, and hence not detectable. Of the 204 undetected faults, 183 were classified as indistinguishable. The 21 remaining faults were not proven to be detectable, but there was insufficient evidence to classify them as never detectable, therefore they were also included in F_α.

Table IV gives the coverage factors for the MATMUL program and also combines the results with those of LINCON in order to get a better feel for coverage during a typical voting frame. Individually, each program detected ≈75 percent of the injected faults; however, the two programs combined detected over 80 percent of the injected faults. Furthermore, 1193 of 1741 faults produced errors in every execution.

Table IV. Coverage Factors—MATMUL

	D_Σ	D_Σ/F	D_Σ/F_α
MATMUL	1336	0.767	0.858
MATMUL∪($\bigcup_{j=1}^{14}$ Run$_j$)	1439	.827	.924
MATMUL∪Run$_1$	1429	.821	.917
MATMUL∩($\bigcap_{j=1}^{14}$ Run$_j$)	1193	.685	.766
MATMUL∩Run$_1$	1238	.711	.795

Discussion. Recognizing that these two programs do not fully exercise the hardware, and that their execution time in real terms is approximately 2 ms, if the experiment were expanded to incorporate a complete voting frame including operating system overhead, it is likely that the detection probabilities would increase further. However, the amount of computation time required for this small sample was prohibitive. Each of the 14 runs of the LINCON program required ≈100 hours of MicroVAX II cpu time. The MATMUL simulation (for fault set F) required ≈25 cpu hours on a MicroVAX 3200. Fortunately, it was possible to distribute the computation requirements across a 16-node network in a batch environment, thus allowing for near linear speedup of the computation time required. Submitting the simulation tasks in low-priority batch mode also allowed potential for completing much of the simulation during periods of low resource utilization.

Another limiting factor is that the simulated processor is small by today's standards. The SS-1750A used in this study consisted of approximately 3500 gates. Current generation microprocessors consist of hundreds of thousands of gates. Thus it is impractical to use this simulation strategy to estimate comparison monitoring coverage parameters.

Coincident Error Measurement

The results of the LINCON simulation were analyzed again, this time in an attempt to corroborate results from the S-GGLOSS experiment measuring coincident error. The LINCON program is more complicated than the simple program used in the S-GGLOSS study, but it does have a similar structure. The program used for the S-GGLOSS study was a simple loop consisting of 10 instructions. There were no branch instructions within the body of the loop, thus every instruction in the program was executed in each iteration. The LINCON program, while still a simple example, exhibits more characteristics of a typical program. Within its main loop are conditional branches and internal loops. The section of code executed in any given iteration is more dependent upon the data than was the case in the S-GGLOSS study. However, as can be seen by referring back to table III, typically 85 percent of the faults detected by this program were detected in the first iteration (D_o/D_Σ). Thus, irrespective of the data, the majority of faults detectable by a given program will produce erroneous behavior in the first iteration.

Discussion of prior results. S-GGLOSS was used to simulate a 300-gate "mini-microcomputer" configured in a simple triplex fault-tolerant architecture (ref. 6). The simulated system was configured as a simple flight controller. The inputs are assumed to be uncorrelated variations in flight path due to mild turbulence. Each identical channel outputs its computed values to an assumed perfect voter/monitor that in turn drives a control surface actuator. The voter/monitor has the capability of detecting and isolating all single-channel errors while masking the failure with the voter. The monitor can also detect three different channel values and transfer control to a backup unit. Thus the only way the monitor can be defeated is when it receives two identical incorrect channel values. Given that two faults (one latent) exist in distinct redundant channels of a fault-tolerant digital system, $\bar{\delta}$ is the probability that they produce identical errors. The S-GGLOSS method for determining coincident error $\bar{\delta}$ is described in Swern et al. (ref. 6).

This factor $\bar{\delta}$ was combined with an average latency measure to determine the contribution of

coincident latent faults to system unreliability. This average latency measure was estimated to be 4.2 iterations, where an iteration corresponded to a single pass through the simulated program. An iteration of a typical flight control program was assumed to last 100 ms, for an average latency of 420 ms. Using these values, the estimated contribution to probability of system failure was $\approx 10^{-11}$ for a 1-hr flight (ref. 6, p. 1004). However, considering that a single iteration consisted of 10 instructions, with each instruction requiring 5 clock cycles to complete, an iteration in the S-GGLOSS study represents 50 clock cycles. If we assume a clock rate of 1 MHz, the time required for a single iteration is 50 μs. Thus, while the average latency time was measured to be 4.2 iterations in the S-GGLOSS study, the extrapolation to an iteration duration of 100 ms is unrealistic, as this assumes that a 50 μs task is the only application in a 100 ms voting frame. In more realistic settings, several applications run consecutively in each voting frame. Therefore, the average latency time for a fault will be much reduced. The only valid conclusion concerning latency would be that for multiple consecutive executions of this 50 μs task, the average latency time would be $4.2 \times 50\ \mu s = 210\ \mu s$. Thus, these results tell us nothing about the behavior of longer term latent faults. One common thread among all of these studies is that for a sufficiently complex program, a significant majority of the faults observed to be excitable by that program are detectable following the first execution of the program.

Results and analysis. The S-GGLOSS study estimated that coincident error $\bar{\delta}$ occurred in 7 percent of the cases. The data generated by the simulations of the LINCON program were analyzed in order to make a similar measurement. In order to be as consistent as possible with the previous study, only the results of the first iteration were considered in measuring $\bar{\delta}$. The sets of detected faults correspond to D_o from table III. Of the 1306 faults ever detected in the first iteration of the program (column D_o, row $\bigcup_{j=1}^{14} \text{Run}_j$), 392 were sometimes latent. Computing in the same fashion as done in the S-GGLOSS study, $\bar{\delta}$ was measured to be 11 percent. This result is consistent with the 7 percent reported in the S-GGLOSS study.

However, upon analysis of the errors produced, it was observed that one error pattern was significantly more frequent than any other. Analysis of the SS-1750A architecture revealed that the dominant error pattern corresponds to the inability of the processor to produce an answer (i.e., the fault causes the processor to lose control).

The interesting point is that a nonanswer does not require a comparison monitor for detection. It can be detected simply by determining if the output register has been written. For example, when the comparison monitoring system gets data, it clears a bit in an output-status register. When a processing element produces new data to place in the output register, it resets this bit. In the next voting frame, if the comparison monitor executive sees that this bit has not been reset, it knows the data in the register are invalid.

Although it was not practical to alter the simulation of the SS-1750A in this fashion, the effect on $\bar{\delta}$ can be measured by excluding the nonanswers from the analysis. Table V shows the proportion of nonanswers N_1 produced during the first iteration of the LINCON program. In each of the 14 runs, at least one of the latent faults produced a nonanswer in the first iteration of a different run. Thus, included in the computation for $\bar{\delta}$ were several instances of faults that produced errors coincident with $\approx$70 percent of the faults detected in the first iteration. If these faults are excluded from consideration, the estimate for $\bar{\delta}$ becomes 1.1 percent.

Table V. Proportion of Nonanswers—LINCON (First Iteration)

LINCON	D_o	N_1	N_1/D_o	N_1/F	N_1/F_α
Run_1	1227	811	0.661	0.466	0.521
Run_2	1074	783	.729	.450	.503
Run_3	1029	785	.763	.451	.504
Run_4	1103	807	.732	.464	.518
Run_5	1182	790	.668	.454	.507
Run_6	1097	794	.724	.456	.510
Run_7	1110	781	.704	.449	.501
Run_8	1056	772	.731	.443	.496
Run_9	1184	787	.665	.452	.505
Run_{10}	1183	786	.664	.451	.504
Run_{11}	1062	794	.748	.456	.510
Run_{12}	1155	813	.704	.467	.522
Run_{13}	1160	791	.682	.454	.508
Run_{14}	1249	797	.638	.458	.512

Another potential source for error in the estimate of $\bar{\delta}$ is that the measurement only considers a single word of voted data. In a typical control system, several different functions are computed within a voting frame, thus more than a single word of data is voted in each frame. If we treat the eight passes through the LINCON program as a single function that produces 16 words of data (the primary output for each pass through the program and an additional

8 words of scratch pad space), the vote can be treated as a block vote of 16 words. The proportion of nonanswers in this scenario is given in table VI. None of the faults that produced a nonanswer were considered latent by the above definition, so no steps were required to account for them in the estimate of $\bar{\delta}$. In this scenario the estimate for $\bar{\delta}$ was measured to be 0.36 percent. This implies that coincident error becomes less of a concern if the vote function encompasses a large enough set of data and is also capable of detecting a nonanswer.

Table VI. Proportion of Nonanswers—LINCON

LINCON	D_Σ	N_Σ	N_Σ/D_Σ	N_Σ/F	N_Σ/F_α
Run_1	1331	779	0.585	0.447	0.500
Run_2	1302	755	.580	.434	.485
Run_3	1319	767	.582	.441	.492
Run_4	1321	787	.596	.452	.505
Run_5	1311	744	.568	.427	.478
Run_6	1290	762	.593	.438	.489
Run_7	1298	743	.575	.427	.477
Run_8	1320	756	.573	.434	.485
Run_9	1306	741	.567	.426	.476
Run_{10}	1328	751	.566	.431	.482
Run_{11}	1310	766	.585	.440	.492
Run_{12}	1325	786	.593	.451	.504
Run_{13}	1289	756	.587	.434	.485
Run_{14}	1328	773	.582	.444	.496

Concluding Remarks

The initial test of GGLOSS proceeded in two distinct phases. The first phase compares results obtained from GGLOSS simulations with those observed in hardware. In the first few comparisons of fault-free behavior there were several observed discrepancies. However, most were caused by misinterpretation of the processor documentation. There were also some difficulties encountered by inadvertently violating some of GGLOSS's underlying assumptions. Similarly, the incomplete microprocessor design caused additional problems. These were all resolved and a good fault-free simulation was eventually obtained.

This made it possible to compare self-test results while injecting stuck-at faults in the combinatorial logic of the microprocessor's ALU. It was possible to exploit GGLOSS's simulation strategy to reveal a previously undocumented error in the design of the microcoded self-test routine. Furthermore, comparison to results from the hardware fault insertion demonstrated that GGLOSS correctly models stuck-at faults in combinatorial logic.

While the code implementing the GGLOSS tool was well written, it is not clear that GGLOSS is capable of performing one of its desired functions, namely, that of capturing the behavior of latent faults and their effects on fault-tolerant computing systems. It was possible to recreate results of earlier studies that attempted to capture characteristics of fault behavior in comparison monitoring systems. However, the limited amount of real time simulated in these experiments restricts the conclusions concerning the behavior of latent faults. None of the studies to date have simulated more than a few milliseconds of real time, thus any observed fault behavior corresponds to either nonlatent faults or faults with very short average latency periods. Because of the computational burden required for fault simulation, it is perhaps questionable that one would want to try to capture the behavior of latent faults by simulation.

While the results concerning the behavior of latent faults are less than promising, there are other ways to approach the problem. The most interesting result of the BGLOSS BDX-930 study was the demonstration of a reasonably fast ($\approx$1 ms) high-coverage (97.4 percent) self-test program.[13] This suggests that for analysis of fault-tolerant systems, one need not depend upon coverage factors based upon an estimate of the effectiveness of comparison monitoring, but rather incorporate an effective periodic background self-test as part of the system overhead. This is not to say that comparison monitoring should not be used. In fact, these studies all indicate that a majority of faults propagate quickly, and thus we depend upon the comparison monitoring system to mask any error. Therein lies the key: Comparison monitoring is not a fault detection strategy, but rather an error detection strategy. It is best used to prevent propagation of errors. In order to ensure an appropriate level of fault detection, diagnostic routines are a necessity. Furthermore, microprocessor faults may not be the dominant source of latent faults. It is much more likely that latent faults will be found in memory systems or possibly in redundancy management logic.[14] Therefore, it is probably wiser to focus efforts on developing efficient on-line diagnostics to detect faults in critical circuit locations.

NASA Langley Research Center
Hampton, VA 23665-5225
February 12, 1991

[13] Again excluding bit faults in the PROM.

[14] A possible scenario is given in footnote 9.

Appendix
Application Programs

Code for the LINCON Program

```
 .NAME        LINCON

;VARIABLE DECLARATION

XARRAY:      .EQU        X'A0
YARRAY:      .EQU        X'A9
MARRAY:      .EQU        X'B2
RESULT:      .EQU        R10
TEMPX:       .EQU        R11
TEMPM:       .EQU        R12
K:           .EQU        R13
TEMPK:       .EQU        R14

;END VARIABLE DECLARATION
;-----------------------------------------------------------------

;INITIALIZE VARIABLES
       LIM          K,0
       LIM          TEMPX,0
       L            TEMPM,MARRAY

;END INITIALIZATION OF VARIABLES

;-----------------------------------------------------------------

;BEGIN MAIN PROGRAM LINCON

MAIN:

LOOP1:
      CIM           K,8                  ;FOR K=0 TO 7 DO
      BEZ           END                  ;ELSE DONE AND GOTO END LABEL

      LR            TEMPK,K              ;LOAD R14 TEMPK WITH LOOP COUNT
      AIM           TEMPK,1              ;SO THAT K+1 CAN BE ADDRESSED
      L             TEMPX,XARRAY,TEMPK
                                         ;LOAD X(K+1) INTO TEMPX (R13)
      S             TEMPX,XARRAY,K       ;TEMPX := X(K+1) - X(K)

      LR            RESULT,TEMPX         ;EVALUATION OF
      MSR           RESULT,TEMPM         ;EQUATION
      A             RESULT,YARRAY,K      ;TEMPX * TEMPM + Y(K) = RESULT

      ;BEGIN IF THEN ELSE STATEMENT
      CIM           RESULT,0             ;IF RESULT ≥ 0 THEN
      BGE           DO_RIGHT_HALF        ;GOTO DO_RIGHT_HALF
      BR            DO_LEFT_HALF         ;ELSE GOTO DO_LEFT_HALF
```

```
RETURN:                                          ;RETURN POINT FROM SUBROUTINES
        ;END OF IF THEN ELSE STATEMENT

        L           TEMPM,MARRAY,TEMPK
        MSIM        TEMPM, -1

        AIM         K,1
        BR          LOOP1


END:
      BR          END                   ;INFINITE LOOP TO STOP EXECUTION

;---------------------------------------------------------------
.COMMENT  %
        THIS SUBROUTINE IMPLEMENTS THE RIGHT HALF OF THE LINCON
        FLOWCHART AS GIVEN IN THE BENDIX REPORT %

DO_RIGHT_HALF:

LOOP_RIGHT:

        LR          R0,TEMPM              ;LOAD TEMPM INTO REG.0
        SIM         R0,1                  ;R0 = TEMPM -1
        CIM         R0, -9                ;IF TEMPM -1 = -9 THEN
        BEZ         R_EXIT_1              ;GOTO LABEL R_EXIT_1
                                          ;ELSE BEGIN

        SR          RESULT,TEMPX          ;RESULT = RESULT - TEMPX

        CIM         RESULT,0              ;IF RESULT < 0 THEN BEGIN
        BGE         R1_ELSE               ;ELSE GOTO LABEL R1_ELSE
                                          ;BEGIN IF
        L           R0,YARRAY,K           ;GET YARRAY(K) MOVE INTO R0
        CIM         R0,0                  ;IF YARRAY(K) ≥ 0 THEN
        BGE         R_EXIT_2              ;GOTO LABEL R_EXIT_2
        BR          R_EXIT_1              ;ELSE GOTO LABEL R_EXIT_1

R1_ELSE:
        SIM         TEMPM,1
        BR          LOOP_RIGHT
R_EXIT_1:
        AR          RESULT,TEMPX
R_EXIT_1A:
        ST          RESULT,YARRAY,TEMPK
        ST          TEMPM,MARRAY,TEMPK
        BR          RETURN
R_EXIT_2:
        ST          RESULT,YARRAY,TEMPK
        SIM         TEMPM,1
        ST          TEMPM,MARRAY,TEMPK
        BR          RETURN

;---------------------------------------------------------------
```

```
DO_LEFT_HALF:

LOOP_LEFT:

        LR          R0,TEMPM
        AIM         R0,1
        CIM         R0,9                    ;IF TEMPM +1 = 9 THEN
        BEZ         L_EXIT_1A               ;GOTO L_EXIT_1
                                            ;ELSE BEGIN
        AR          RESULT,TEMPX

        CIM         RESULT,0                ;IF RESULT < 0 THEN
        BLT         L1_ELSE                 ;GOTO L1_ELSE
                                            ;ELSE BEGIN
        L           R0,YARRAY,K
        CIM         R0,0
        BLT         L_EXIT_2
        BR          L_EXIT_1
L1_ELSE:
        AIM         TEMPM,1
        BR          LOOP_LEFT
L_EXIT_1:
        SR          RESULT,TEMPX
L_EXIT_1A:
        ST          RESULT,YARRAY,TEMPK
        ST          TEMPM,MARRAY,TEMPK
        BR          RETURN
L_EXIT_2:
        ST          RESULT,YARRAY,TEMPK
        AIM         TEMPM,1
        ST          TEMPM,MARRAY,TEMPK
        BR          RETURN
;-------------------------------------------------------------------------
.END
```

Code for the MATMUL Program

```
.COMMENT %
1750-A MIL
MATRIX SQUARED (WAS MULTIPLY). ASSUMES THE ARRAY BEING
PROCESSED IS 2 × 2

NOTE MATMUL USES THE FOLLOWING REGISTERS
       R0      - POINTER TO ARRAY A
       R1      - ROW INCREMENT FOR A
       R2      - COUNTER FOR OUTER LOOP(M)
       R3      - COLUMN INDEX FOR ARRAY B
       R4      - COUNTER FOR INNER LOOP(P)
       R5      - OFFSET INTO ARRAY A
       R6,7    - REGISTERS CONTAINING SUM DURING INNER
                      PRODUCT CALCULATION
       R8,9    - RESULT OF MULTIPLICATION DURING
                      INNER PRODUCT CALCULATION
       R10     - INCREMENT FOR ARRAY B OFFSET
       R11     - OFFSET INTO ARRAY B
       R12     - OFFSET INTO ARRAY C
       R13     - COUNTER FOR SUROUTINE LOOP(N)

       AUTHOR:          WILLIAM F. INGOGLY
       CREATED:         7 SEPTEMBER 1985
       MODIFIED BY:     KAREN T. LOONEY
       DATE:            27 AUGUST 1987

       THEN SUBSEQUENTLY MANGLED FOR THIS STUDY
       BY PAUL MINER --- LAST CHANGE: 24 JULY 1989%

.NAME MATRIX_SQUARED
MATMUL:
       LIM          R15,X'00A0'
       LIM          R0,0                ;LOAD POINTER TO ARRAY
       LIM          R1,2                ;INCREMENT FOR ROW
       MIM          R1,2
       LR           R1,R2
       LIM          R2,0
       LIM          R10,2
       MIM          R10,2
       LR           R10,R11
       LIM          R12,0
```

```
LOOP1:
        LIM         R3,0
        LIM         R4,0
LOOP2:
        SJS         R15,INPROD
        AIM         R3,2
        AIM         R4,1
        CIM         R4,2
        BNZ         LOOP2
        AR          R0,R1
        AIM         R2,1
        CIM         R2,2
        BNZ         LOOP1
        BR          HERE        ;END MATMUL
INPROD:
        PSHM        R2,R2
        LR          R5,R0
        LIM         R13,0
        LIM         R6,0
        LIM         R7,0
        LR          R11,R3
LOOP:
        DL          R8,X'0043',R5
        FM          R8,X'004B',R11
        FAR         R6,R8
        AIM         R5,2
        AR          R11,R10
        AIM         R13,1
        CIM         R13,2
        BNZ         LOOP
        DST         R6,X'0053',R12
        AIM         R12,2
        POPM        R2,R2
        URS         R15
.end
```

References

1. Nagel, Phyllis M.: *Modeling of a Latent Fault Detector in a Digital System.* NASA CR-145371, 1978.

2. McGough, John G.; and Swern, Fred L.: *Measurement of Fault Latency in a Digital Avionic Mini Processor.* NASA CR-3462, 1981.

3. McGough, John G.; and Swern, Fred L.: *Measurement of Fault Latency in a Digital Avionic Mini Processor—Part II.* NASA CR-3651, 1983.

4. McGough, John G.: *Feasibility Study for a Generalized Gate Logic Software Simulator.* NASA CR-172159, 1983.

5. McGough, J. G.; and Nemeroff, S.: *The Development of an Interim Generalized Gate Logic Software Simulator.* NASA CR-177939, 1985.

6. Swern, Frederc L.; Bavuso, Salvatore J.; Martensen, Anna L.; and Miner, Paul S.: The Effects of Latent Faults on Highly Reliable Computer Systems. *IEEE Trans. Comput.*, vol. C-36, no. 8, Aug. 1987, pp. 1000–1005.

7. Lala, Parag K.: *Fault Tolerant and Fault Testable Hardware Design.* Prentice-Hall, Inc., c.1985.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-4251		

4. Title and Subtitle	5. Report Date
Test and Evaluation of the Generalized Gate Logic System Simulator	April 1991
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Paul S. Miner	L-16822
9. Performing Organization Name and Address	10. Work Unit No.
NASA Langley Research Center Hampton, VA 23665-5225	505-66-21-02
	11. Contract or Grant No.
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Washington, DC 20546-0001	Technical Memorandum
	14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

This paper discusses the results of the initial testing of the Generalized Gate Logic System Simulator (GGLOSS). The simulator is a special-purpose fault simulator designed to assist in the analysis of the effects of random hardware failures on fault-tolerant digital computer systems. The testing of the simulator covers two main areas. First, the simulation results are compared with data obtained by monitoring the behavior of hardware. The circuit used for these comparisons is an incomplete microprocessor design based upon the MIL-STD-1750A Instruction Set Architecture. In the second area of testing, current simulation results are compared with experimental data obtained using precursors of the current tool. In each case, a portion of the earlier experiment is confirmed. The new results are then viewed from a different perspective in order to evaluate the usefulness of this simulation strategy.

17. Key Words (Suggested by Author(s))
Fault simulation
Fault tolerance
Self-test
Comparison monitor
Coverage

18. Distribution Statement
Unclassified—Unlimited

Subject Category 66

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	19	A03